جامعة بيرزيت

BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Electrical & Computer Engineering Department

Advanced Digital Design ENCS3310

Course Project

Prepared by:

Name: mahmoud nobani **Id:**1180729

Instructor: Dr. Abdallatif Abuissa

Section: 2

Date:20/12/2021

Abstract:

The goal of this project was to build an 8bit signed comparator in two different ways, the first one is by using an 8-bit ripple subtractor and the second one is by using a magnitude comparator, we intend to see how different ways of implementing the same circuit can affect the overall performance, and how can a various delay in the basic logic gates can derange and change our overall design and connections.

For this project I used active VHDL tool.

Contents

Table of figures

Theoretical Design

An n-bit comparator as the name suggests is a combinational logic circuit that aims to compare two different binary numbers giving use a 3-bit result of either greater/smaller/equal.

There are various ways to implement an n-bit comparator, this time we will focus on two ways only as follow:

1) Subtraction

Subtraction is the most basic method to implement a comparator, in theory subtracting A-B, will give us one of three outcomes, either a negative number (thus b>a), positive (thus a>b) or zero (thus a-b).

Now the question how can we implement this using logic circuits?

when we look at the origin of the subtraction process, we can see that it’s in its essence an addition one, basically A-B => A+(-B).

so, to build a subtractor, u basically build an adder, and to build an adder we need to go to its roots which is a 1bit full adder:

a. Full adder

A full adder is the most basic form of addition there is, it consists of three inputs A, B, Carry In

And two output Sum, Carry out.

The logical expression of each one of the outputs is:

Sum = A xor B xor Carry In.

Carry out = A and B or A and Carry in or B and Carry out.

Combining these logical gates together gives us the following block diagram:

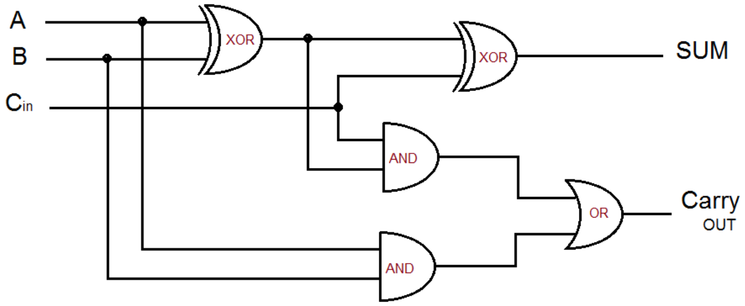


Figure 1: full adder

But the full adder just adds one-bit numbers, which is not efficient, to face this issue we combine multiple full adders together, one famous combination of full adders is ripple adder.

b. Ripple carry adder

Ripple carry adder is a combinational circuit made of multiple full adders in cascade with each other with each carry is “carried/ripple” to the next full adder and so on, the ripple adder can be used to sum any n-bit numbers.

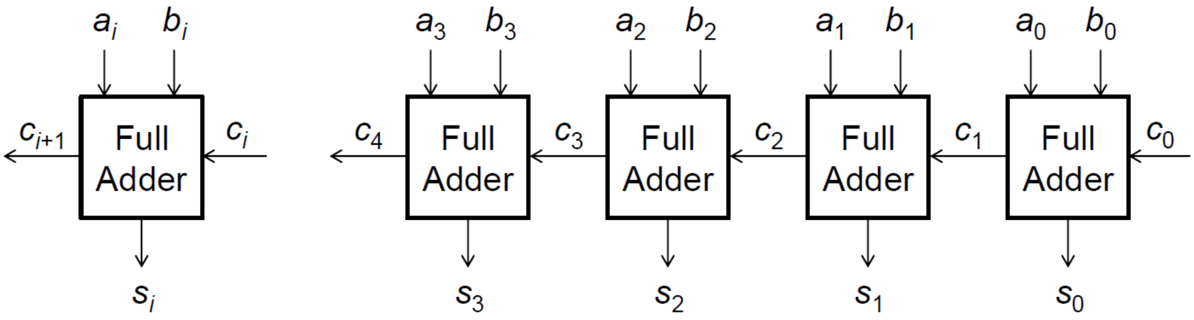


Figure 2:Ripple Carry Adder.

Now after we had a n-bit adder, we need to make it subtractor.

c. Subtractor

A subtractor as explained earlier works the same as an adder, but with the other number in negative, thus to subtract b from a, we make b negative, and negative numbers are represented as 2’comp in binary.

Negative b = 2’comp of b = 1’comp of b +1 = b xor 1 + 1

now after we made a subtractor we need to use it as a comparator.

d. Ripple adder comparator

A comparator has three main outputs, **a greater than b**, **b greater than a**, **a equal to b**, and as we established a subtractor can either give a positive, negative or a zero for a result, so to check which number is bigger a or b we check first the sum if zero, second the sign of the sum if positive or negative.

Checking if the sum is zero or not can be done using nor gate, as nor gives 1 only if its inputs are zeros

Equal = (sum(n) nor sum(n-1)) and (sum(n-2) nor sum(n-3)) and …. nor sum (0).

Now to find out which number is bigger we will look at the sign of the sum, so for a-b, if a is bigger the sign is positive (0), if not then negative (1), thus:

A > B = not sum(n).

Now to avoid the conflict with equals we will use the and gate with equal inverse, thus:

A > B = not sum(n) and not equal

but now we will face another issue, which is overflow,

overflow occurs only when u subtract signed numbers, as the sign will be lost because of the overflow, we can see this the figures below:

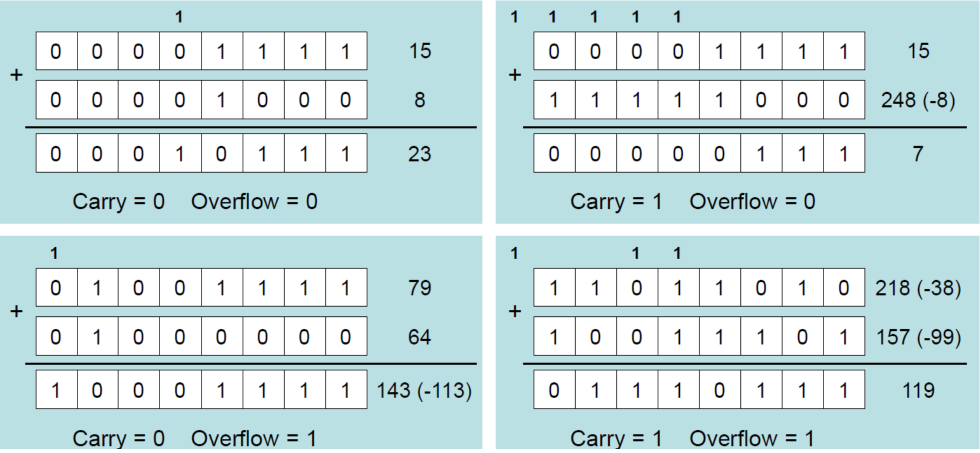


Figure 3: overflow demonstration.

In case 3 and 4 we can see how the sign has changed because of the overflow, so when there is and overflow (overflow = 1), the sign will be different from its original state, so if a > b, and there is an overflow then sum result will be negative instead of positive, so in this case to avoid this problem we will xor the sign bit with the overflow as shown:

A > B = (not sum(n) xor overflow) and not equal.

Note: **overflow =** carry of the most significant bits **xor** carry of the second most significant bits.

This way we solved the overflow issue, and now for the last case, when b > a, and this one is so easy as it only occurs if a > b is zero and equal is zero, thus:

B > A = not A>B and not equal

This way we can successfully implement a comparator from a subtractor.

As for the second way to implement a comparator, we will use the magnitude comparator circuit.

2) Magnitude comparing

Magnitude comparing in theory works when u compare each bit in lets say A, with the corresponding bit in B, so if A=010 and B=011, we will compare A(2)=0 and B(2)=0, then go to A(1)=1 and B(1)=1, and lastly A(0)=0 and B(0)=1, and check each one of these cases, so to be able to do that, we need at first to make 1bit magnitude comparator.

a. 1bit magnitude comparator

To know if **a is bigger than b** we **and a with inverse b**, and to know **if b is bigger than a** we **and b with inverse a**, and lastly to know if **a equals b**, we just take **xnor** of the **last two results**, as shown in the figure below:

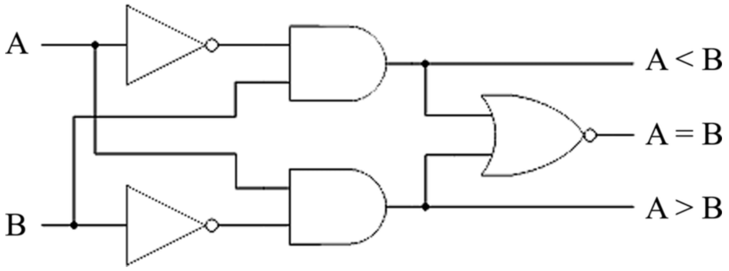


Figure 4: 1bit magnitude comparator

Using this model, we will be able to implement an n-bit comparator.

b. N-bit comparator

At first, we will need N chips of 1bit comparator, and to find the first output which is if the two numbers are equals, we will just use and gate with the outputs of xnor as shown:

Equal = e (0) and e (1) …. And e(n).

Where e(n) is whether a(n) and b(n) are equals**.**

Now for the case of a>b, and here were things gets a little bit tricky. So, in theory we need to compare the most significant bit first, if they are not equal, we go to the second, and so on.

But how we can keep with them if they are not equal?

The trick is we and the result of the bit with the previous bit result of equality, so basically, we compare the most significant bit first, then or with the equal of the most significant bit and compare with result of next bit and so on, this idea can be written as a logical expression as follow:

a>b = a>b(n) or (e(n) and a>b(n-1)) or (e(n) and e(n-1) and a>b(n-2)) ……

so, in theory if a=011 and b=010, it will compare the values giving this result:

a>b = 0 or (1 and 0) or (1 and 1 and 1) = 1, which is true.

But there is a small trick here, which is a and b are signed, thus last bit is a signed one (0 for positive, 1 for negative)

So basically, if a(n) = 0 and b(n) = 1, the 1bit comparator will give us the b > a, but a is bigger in this case, so to avoid this, I edited the last equation to be like this:

a>b = b>a (n) or (e(n) and a>b(n-1)) or (e(n) and e(n-1) and a>b(n-2)) ……

so, I just replaced the result of comparing most significant bit (a(n)>b(n)) with (b(n)>a(n)) and hence the issue is solved.

Lastly to find if b>a, we do the following:

B > A = not A>B and not equal

Now for the last part of the design, the testbench.

3) Testbench:

The testbench main purpose as the name suggests, is to test if the circuit is working or not, it mainly consists of three components, the first is a test generator , that generates test inputs for the circuit, I made a test generator that generates outputs between -128 and 127 for each number, the testbench will also find the exact value using if statements and simple high level logic, the other components the testbench, is the circuit under test, which will take the test inputs and give us the output that we will check, and lastly the result analyzer, which will take the output of the circuit and the output of the test generator and check if it will give right values or not.

One extra note about the circuit, we will use d-flip flops, to take the outputs out of it, for two reasons, the first is to avoid glitches if exists, and the second is for the circuit to keep up with the result analyzer and the test generator, because of the delays in the logic gates.

Design procedure and results:

Subtraction Comparator:

After the theoretical design was established, now we start with the real design, as for the code, u can check the **Appendix**Appendix, in here I will show the block diagram which will show the main components, starting with the adder/subtractor, check the figure below:

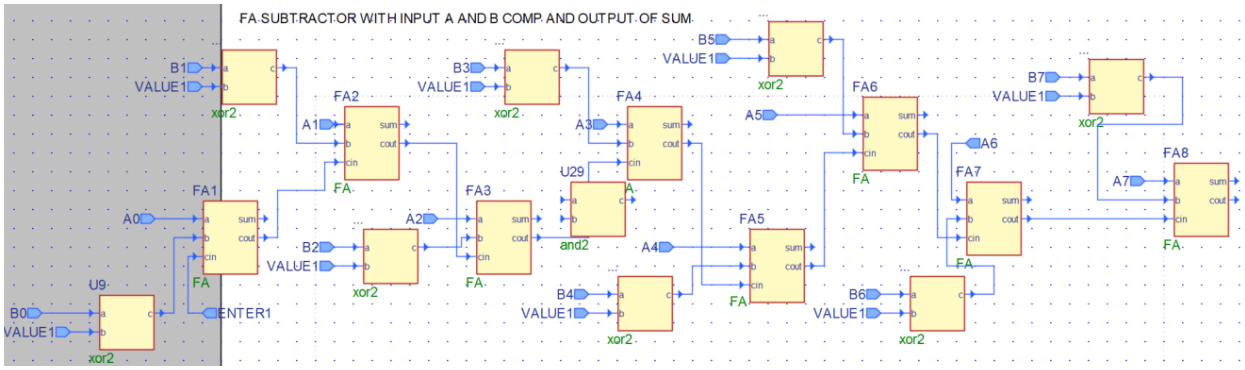


Figure 5: subtractor block diagram

The design here is simple, I built 8 FA, combined them in a ripple way with each other, and for the 1'comp of b I just xor each b of b with one, then made it the input for the adder, and lastly to make it 2'comp, I made the first carryin 1, and that's it for the adder, now for how made the comparator, starting with the equal procedure shown in the figure below:

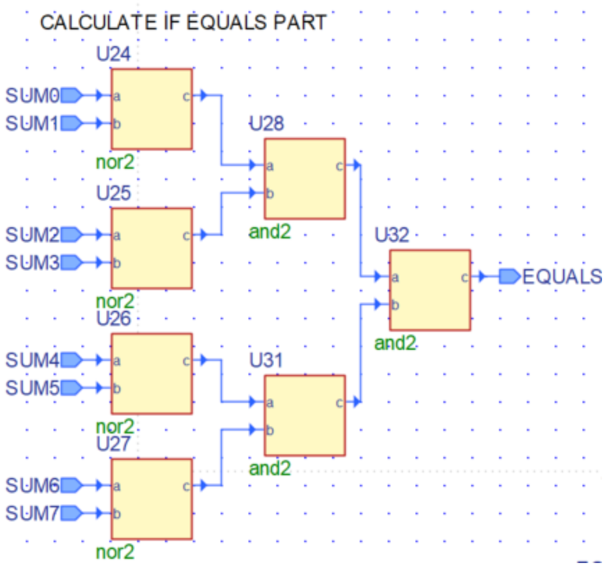


Figure 6:subtraction comparator equals procedure

As for the sum, as explained earlier, we use a combination of nor and gate with all bits in the sum, which will give us the indication if the numbers are equal or not.

Next is a>b, and the block diagram for it is shown below:

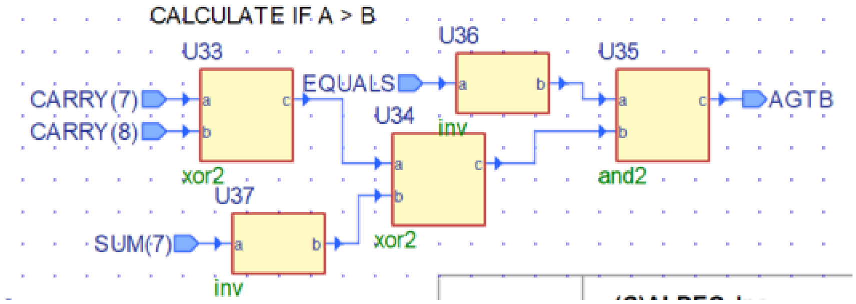


Figure 7: subtraction comparator a>b

This now is simple too, following what was established in the theory, I found if the design had an overflow or now, then used it with xor gate and the inversion of the sign and lastly and with the inversion of equals, this produced the combinational circuit above, lastly for b>a, I just used and with the inversion of equal and a>b, as shown in the figure below:

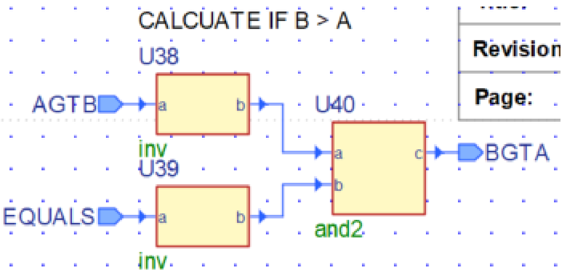


Figure 8:subtraction comparator b>a

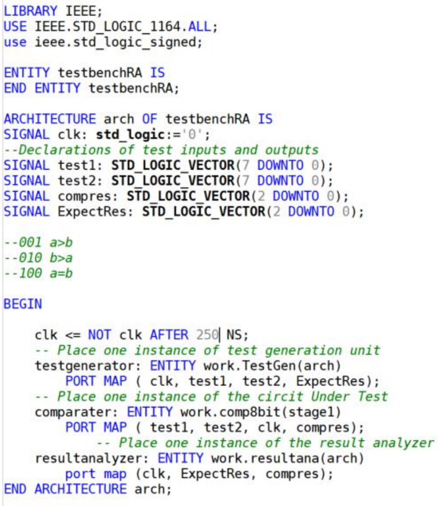
Now before we start checking the results, we need to take notice of the various delays that are present in the simple logic gates as shown below:

Gate	Delay
Inverter	2 ns
NAND	5 ns
NOR	5 ns
AND	7 ns
OR	7 ns
XNOR	9 ns
XOR	12 ns

Figure 9: delays

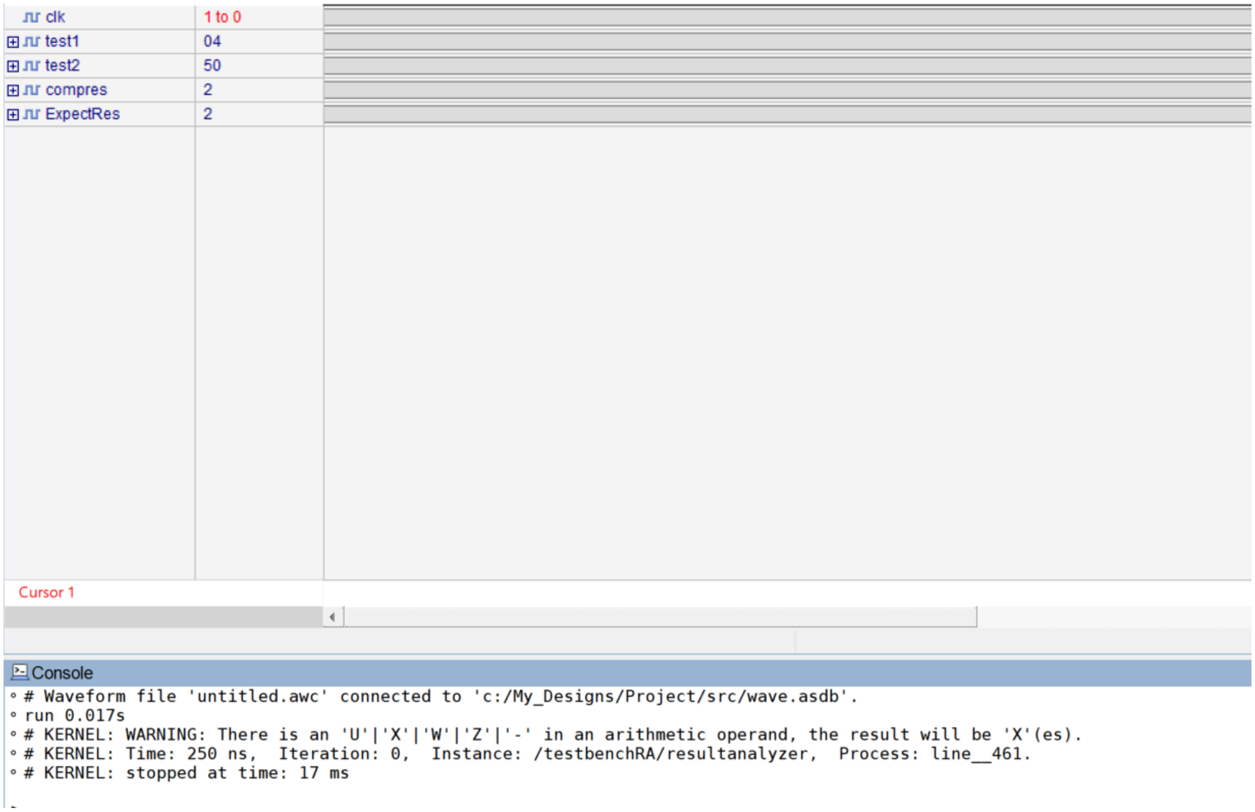
These delays will introduce a latency in the circuit, to find out what is the latency for each circuit we ran a simulation on each stage, taking in count only and only the extreme cases, so in the case stage1 (comparator made from a ripple adder), **A will have a value of FE, and B will have a value of FF**, this will introduce a latency of **212 ns, which is the maximum latency**, thus for the clock of the system I used a clock with 250 ns clock cycle, with simple math I have 256*256 test cases, multiple by the clock cycle will give me 0.017 s overall period to check all the cases

running the testbench code below will give the following results:



```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_signed;

ENTITY testbenchRA IS
END ENTITY testbenchRA;

ARCHITECTURE arch OF testbenchRA IS
SIGNAL clk: std_logic:='0';
--Declarations of test inputs and outputs
SIGNAL test1: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL test2: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL compres: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL ExpectRes: STD_LOGIC_VECTOR(2 DOWNTO 0);

--001 a>b
--010 b>a
--100 a=b

BEGIN

    clk <= NOT clk AFTER 250 NS;
    -- Place one instance of test generation unit
    testgenerator: ENTITY work.TestGen(arch)
        PORT MAP ( clk, test1, test2, ExpectRes);
    -- Place one instance of the circit Under Test
    comparater: ENTITY work.comp8bit(stage1)
        PORT MAP ( test1, test2, clk, compres);
            -- Place one instance of the result analyzer
    resultanalyzer: ENTITY work.resultana(arch)
        port map (clk, ExpectRes, compres);
END ARCHITECTURE arch;
```

Figure 10: testbench for subtraction comparator

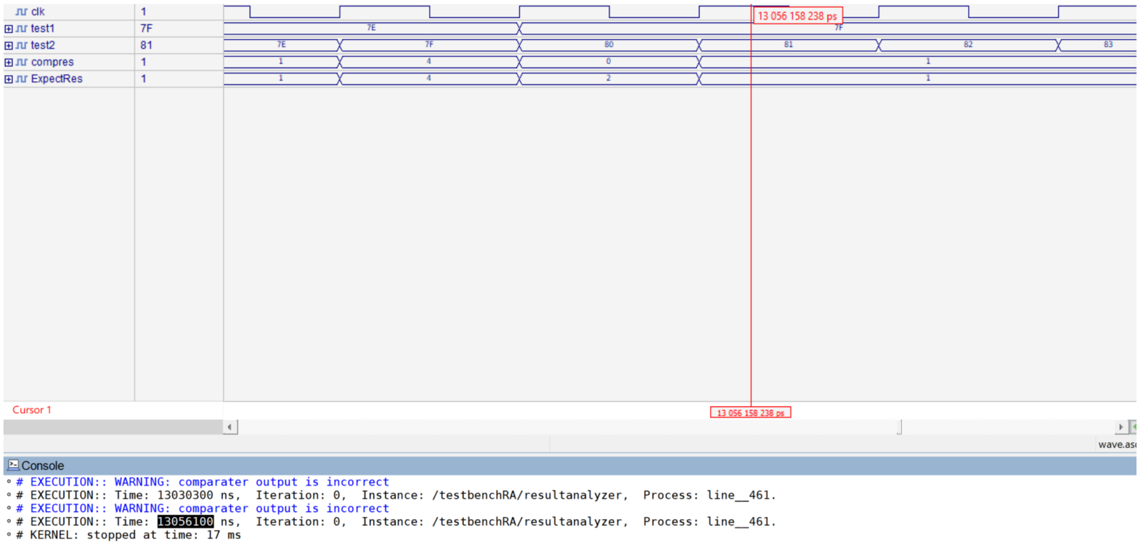


```
# Waveform file 'untitled.awc' connected to 'c:/My_Designs/Project/src/wave.asdb'.
run 0.017s
# KERNEL: WARNING: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
# KERNEL: Time: 250 ns,  Iteration: 0,  Instance: /testbenchRA/resultanalyzer,  Process: line__461.
# KERNEL: stopped at time: 17 ms
```

Figure 11: subtraction comparator, results of test1 (no errors)

There isn’t much present in the waveform, but we didn’t get any assert waring from the result analyzer which confirms the comparator is working fine.

Now if we change the clock cycle to 100 (which is less than the latency), that will give as the following results:

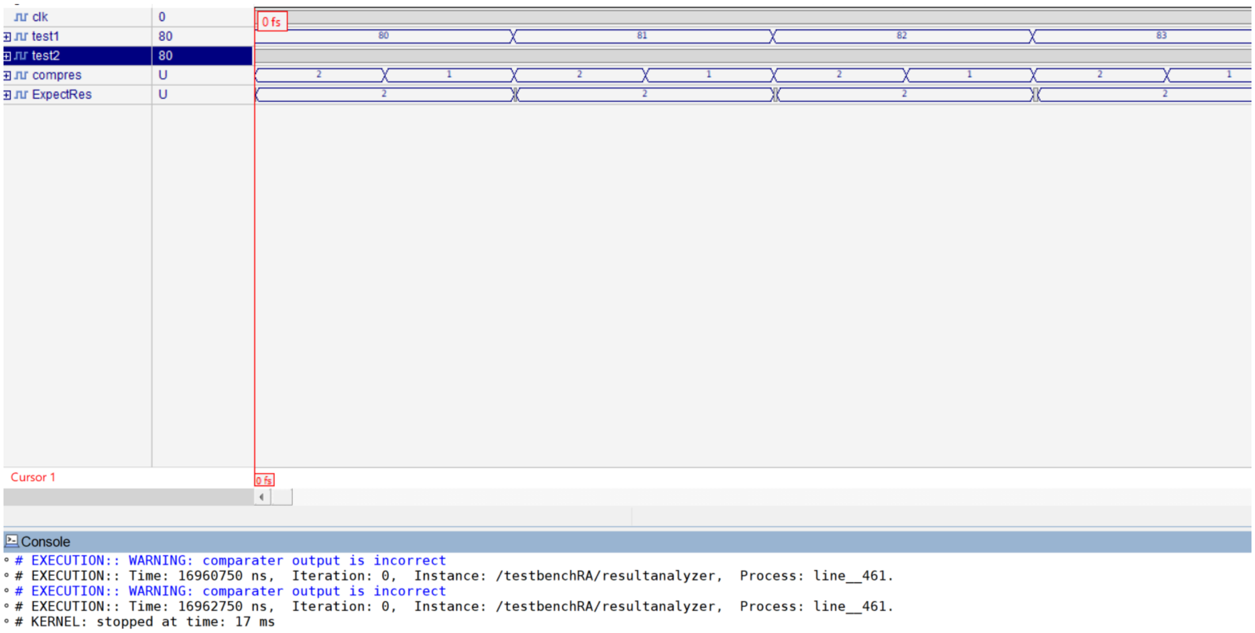


```
# EXECUTION:: WARNING: comparater output is incorrect
# EXECUTION:: Time: 13030300 ns,  Iteration: 0,  Instance: /testbenchRA/resultanalyzer,  Process: line__461.
# EXECUTION:: WARNING: comparater output is incorrect
# EXECUTION:: Time: 13056100 ns,  Iteration: 0,  Instance: /testbenchRA/resultanalyzer,  Process: line__461.
# KERNEL: stopped at time: 17 ms
```

Figure 12: subtraction comparator, results of test2 (with errors)

We can see how did the new clock affect the output as it gave Nemours warnings, and the case in the figure show some inconsistency in the waveform.

And now for the last test, we will bring back the old clock cycle, and now instead we will introduce a different type of error, I will change the sum architecture inside the full adder to have or instead of xor, this will give us the following results:



```
# EXECUTION:: WARNING: comparater output is incorrect
# EXECUTION:: Time: 16960750 ns,  Iteration: 0,  Instance: /testbenchRA/resultanalyzer,  Process: line__461.
# EXECUTION:: WARNING: comparater output is incorrect
# EXECUTION:: Time: 16962750 ns,  Iteration: 0,  Instance: /testbenchRA/resultanalyzer,  Process: line__461.
# KERNEL: stopped at time: 17 ms
```

Figure 13: subtraction comparator, results of test3 with errors.

We can see in here the multiple inconsistency introduced by changed the sum equation from xor to or.

And as we saw we can conclude that both the comparator we made, and the test bench works perfectly, now for the other part, the comparator made from magnitude comparator:

Magnitude Comparator:

Starting with the block diagram, I made 8 chips of 1bit comparator, which will compare each bit in a with the corresponding one in b, and give if it greater, less, or equal, the combination is as shown below

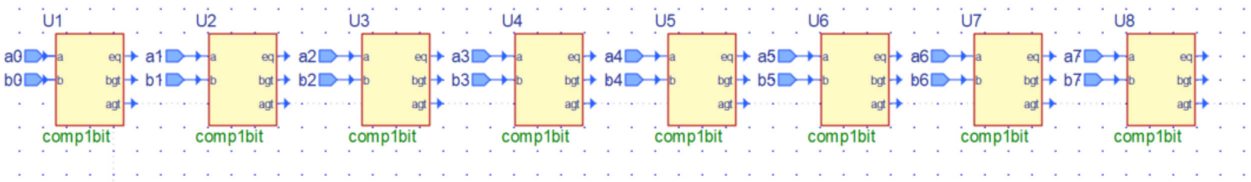


Figure 14:magnitude comparator, 8bit to bit comparator

Now, how we made equals, its simple, just a combination of and gate as shown below:

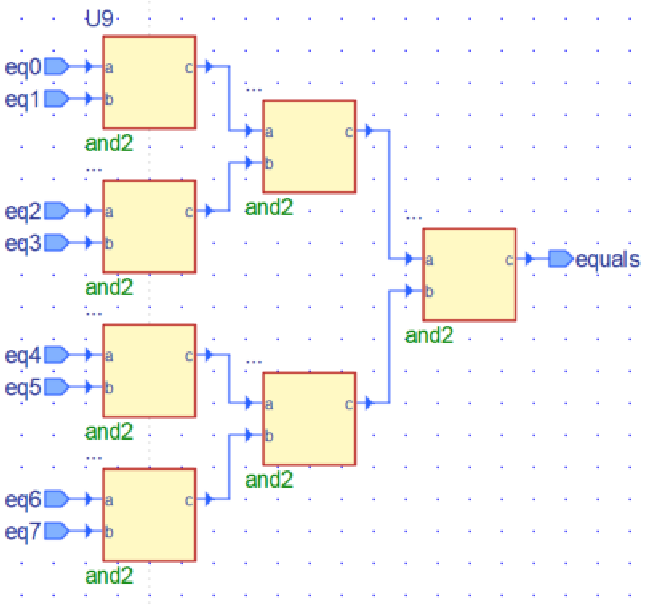


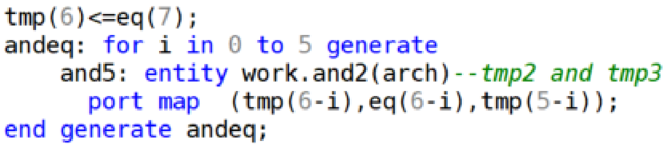
Figure 15:magnitude comparator, equals

as for a>b, and this part is really tedious to implement with simple logic gates, so I will show how I was able to implement it using VHDL logic.

As established earlier, the equation for a>b is the following:

a>b = b>a (n) or (e(n) and a>b(n-1)) or (e(n) and e(n-1) and a>b(n-2)) ……

every time the bit is smaller, the implementation becomes more and more tedious, to avoid this I use generate command, to give me a vector that has the values of equals with and gate, the code in the figure below:

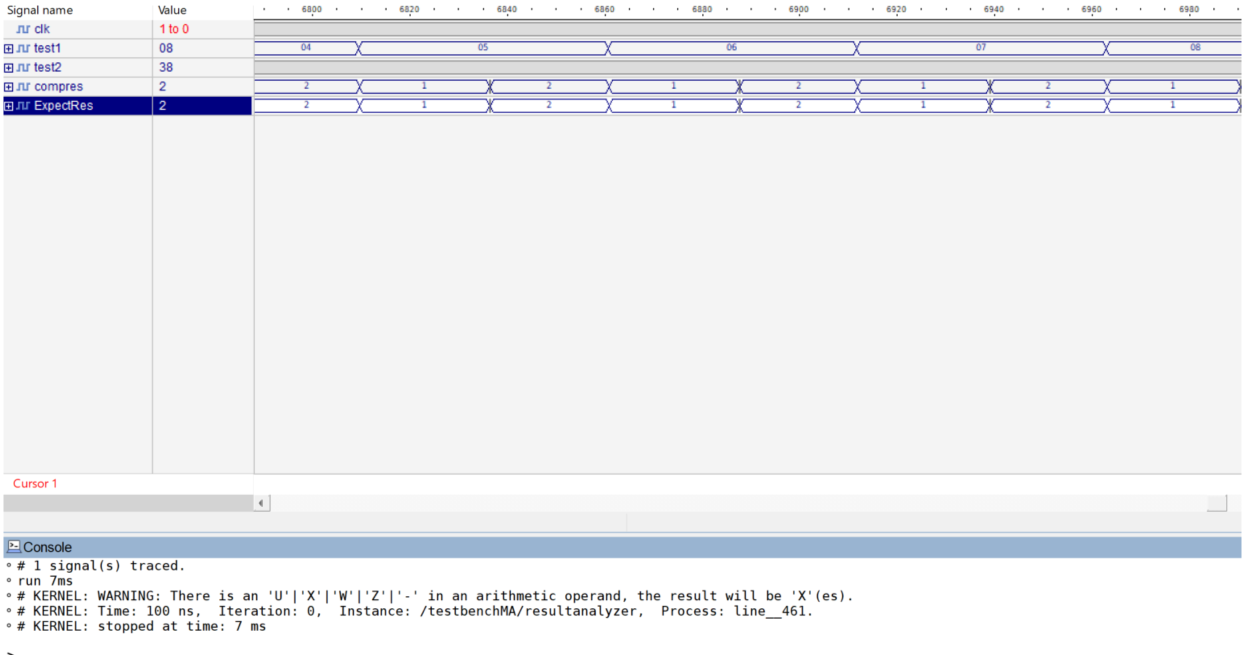


```
tmp(6)<=eq(7);
andeq: for i in 0 to 5 generate
    and5: entity work.and2(arch)--tmp2 and tmp3
      port map  (tmp(6-i),eq(6-i),tmp(5-i));
end generate andeq;
```

Figure 16:magnitude comparator, vector of e anding

After we now have the values of e, we can now just and it with the values of a>b for each corresponding bit, and lastly use or, this is also done using generate.

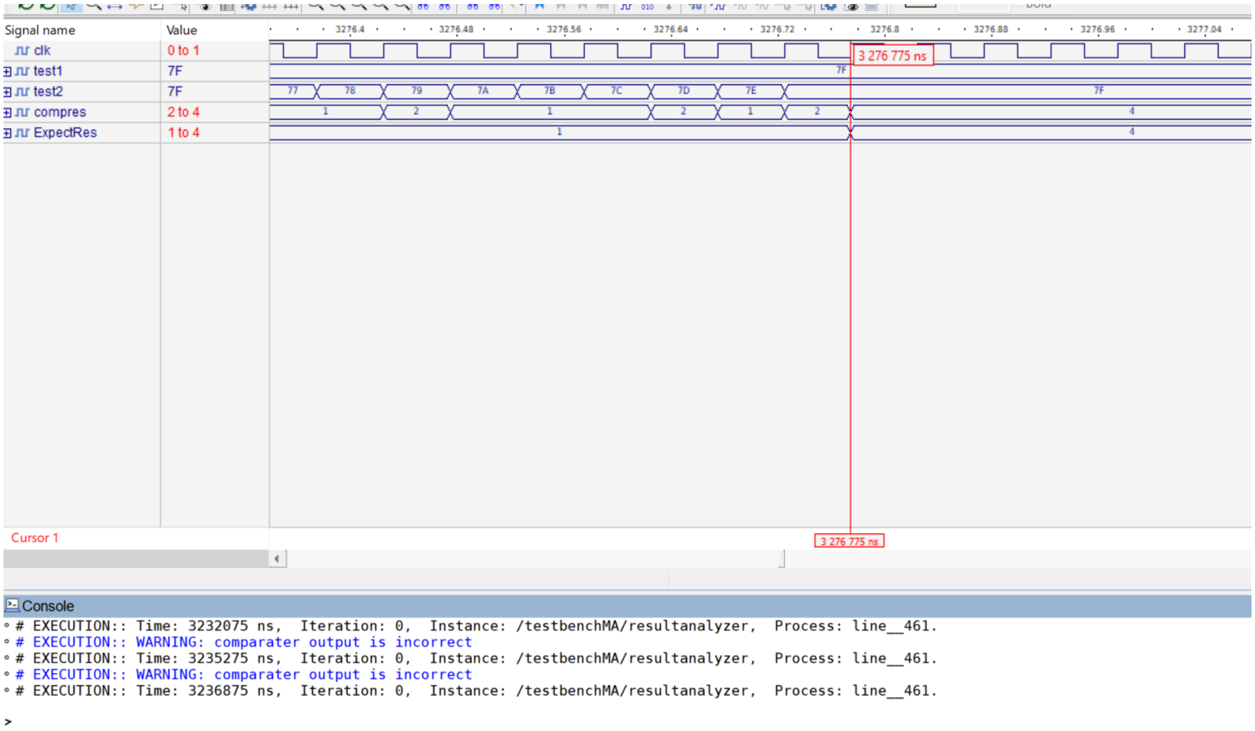
As for the results, The same way we started with the previous one, we will find the latency first, and it can be found by giving a worst cast input, in this case the worst case will be a = 01 and b = 00, in this case the result will be shown after 97ns, thus the latency, to avoid it I introduced a clock cycle of 100 ns, which gave us the following results:



```
# 1 signal(s) traced.
run 7ms
# KERNEL: WARNING: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
# KERNEL: Time: 100 ns,  Iteration: 0,  Instance: /testbenchMA/resultanalyzer,  Process: line__461.
# KERNEL: stopped at time: 7 ms
```

Figure 17: test4 without errors

As we can see no warning were given, thus the run worked just fine, but now how about when we make the clock cycle 25 ns instead of 100ns

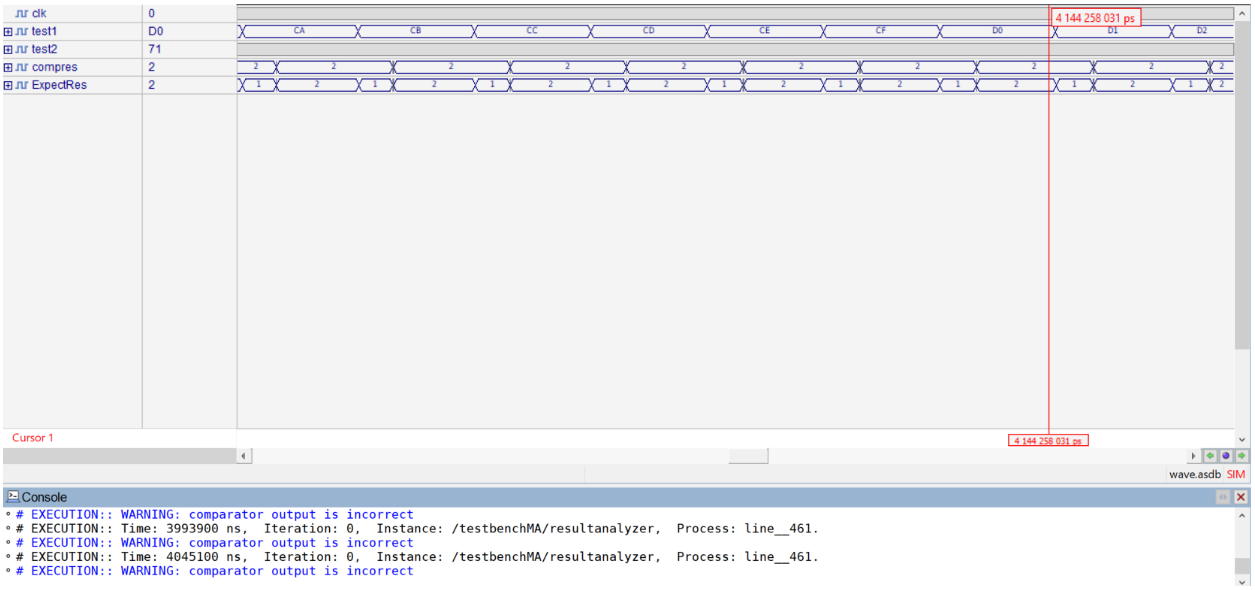


```
# EXECUTION:: Time: 3232075 ns,  Iteration: 0,  Instance: /testbenchMA/resultanalyzer,  Process: line__461.
# EXECUTION:: WARNING: comparater output is incorrect
# EXECUTION:: Time: 3235275 ns,  Iteration: 0,  Instance: /testbenchMA/resultanalyzer,  Process: line__461.
# EXECUTION:: WARNING: comparater output is incorrect
# EXECUTION:: Time: 3236875 ns,  Iteration: 0,  Instance: /testbenchMA/resultanalyzer,  Process: line__461.
>
```

Figure 18:test5 with errors

We can see here how a smaller clock cycle introduced a lot of errors, and lastly for the last test, what if I changed a the or of a>b to and?

The result I got is shown below:



```
# EXECUTION:: WARNING: comparator output is incorrect
# EXECUTION:: Time: 3993900 ns,  Iteration: 0,  Instance: /testbenchMA/resultanalyzer,  Process: line__461.
# EXECUTION:: WARNING: comparator output is incorrect
# EXECUTION:: Time: 4045100 ns,  Iteration: 0,  Instance: /testbenchMA/resultanalyzer,  Process: line__461.
# EXECUTION:: WARNING: comparator output is incorrect
```

Figure 19:test6 with errors

And look at those wrong values I got, there are a lot of wrong values and warning given, this proves that my testbench works perfectly well, and my comparator work well too

Conclusion:

In conclusion we can see, that the two implemented designs worked perfectly, as both of them gave accurate results, but in the case of the subtraction comparator, it needed much higher clock cycle than the other design, which make the second one more effective than the first one, we can also see how useful is the result analyzer and test generator are, as they can calculate millions of case in just seconds, but despite all this we can always improve on the current design by finding more efficient design, maybe in the case of the magnitude comparator, we can find a less tedious logic to get if a>b, by implementing more of an iterative design that has a bigger base e.g 2-bit comparator instead of one, as for the subtractor, maybe we can use another way to implement it instead of using the ripple adder (212 delay), the carry look ahead adder is a good option especially with big bit number ,as it will have less delay (148 delay),, and the carry-select (101) and Manchester adder (56) have even less delays than the look ahead adder, so there are better ways to implement this type of comparator, and there is lots of ways we can improve the performance of my desing.

Appendix

```
--------------------------------------
--dff3bit
--------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dff3 IS
PORT ( d : IN STD_LOGIC_vector(2 downto 0); clk: IN STD_LOGIC;Q : OUT STD_LOGIC_vector(2 downto 0));
END ENTITY dff3;

ARCHITECTURE arch OF dff3 IS
BEGIN
	PROCESS (clk)
	BEGIN
		IF ( rising_edge(clk) ) THEN
			q <= d;
		END IF;
	END PROCESS;
END ARCHITECTURE arch;

--------------------------------------
--inverter with delay 2ns
--------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

```
--USE ieee.std_logic_signed.ALL;

ENTITY inv IS
        PORT ( a: IN STD_LOGIC; b: OUT STD_LOGIC);
END ENTITY inv;

ARCHITECTURE arch OF inv IS
BEGIN
b <= not a after 2ns;
END ARCHITECTURE arch;

-------------------------------------
--nand with delay 5ns
-------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--USE ieee.std_logic_signed.ALL;

ENTITY nand2 IS
        PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY nand2;

ARCHITECTURE arch OF nand2 IS
BEGIN
c <= a NAND b after 5ns;
END ARCHITECTURE arch;
```

```
-----------------------------------
--nor with delay 5ns
-----------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY nor2 is
        PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY nor2;
ARCHITECTURE arch OF nor2 IS
BEGIN
c <= a nor b after 5ns;
END ARCHITECTURE arch;

----------------------------------
--or with delay 7ns
----------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY or2 is
        PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY or2;
ARCHITECTURE arch OF or2 IS
BEGIN
c <= a OR b after 7ns;
```

```
END ARCHITECTURE arch;

------------------------------------
--and with delay 7 ns
------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and2 is
        PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY and2;
ARCHITECTURE arch OF and2 IS
BEGIN
c <= a and b after 7ns;
END ARCHITECTURE arch;

------------------------------------
--xor with delay 12 ns
------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY xor2 is
        PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY xor2;
ARCHITECTURE arch OF xor2 IS
```

```
BEGIN
C <= a xor b after 12ns;
END ARCHITECTURE arch;

---------------------------------------
--xnor with a delay 9ns
---------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY xnor2 is
        PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY xnor2;
ARCHITECTURE arch OF xnor2 IS
BEGIN
c <= a xnor b after 9ns;
END ARCHITECTURE arch;

----------------------------------------
--comparater entity
----------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;

entity Comp8bit is
```

```
	--generic (n: integer :=8);
	port ( a,b: in std_logic_vector(7 downto 0);
	clk: in std_logic;
	output: out std_logic_vector(2 downto 0) );
end entity Comp8bit;

------------------------------------
--stage 1
------------------------------------

--1 Bit FA



LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity FA is
	port ( a,b,cin: in std_logic; sum,cout: out std_logic);
end entity FA;

ARCHITECTURE arch OF FA IS
signal tmp1,tmp2,tmp3,tmp4,tmp5: std_logic;
BEGIN
--sum <= cin XOR a XOR b;
sum1: entity work.xor2(arch)
	port map  (a,b,tmp1);--a xor b
sum2: entity work.xor2(arch)
	port map  (tmp1,cin,sum);--cin xor a xor b
```

```
--end of sum
--cout <= ( a AND b ) OR ( cin AND a ) OR ( b AND cin );
and1: entity work.and2(arch)
        port map  (a,b,tmp2);  --a and b
and2: entity work.and2(arch)
        port map  (cin,a,tmp3); --a and cin
and3: entity work.and2(arch)
        port map  (b,cin,tmp4);--cin and b
or1: entity work.or2(arch)
        port map  (tmp2,tmp3,tmp5);--( a AND b ) OR ( cin AND a )
or2: entity work.or2(arch)
        port map  (tmp4,tmp5,cout);--( a AND b ) OR ( cin AND a ) OR ( b AND cin );
END ARCHITECTURE arch;

-------------------------------------
--stage 1 ripple adder architecter
-------------------------------------


architecture stage1 of Comp8bit is
signal s: std_logic_vector(8 downto 0);--for carries
signal Bcomp: std_logic_vector(7 downto 0);--for comp of b
signal res: std_logic_vector(7 downto 0); --for the result of subtraction
signal equals: std_logic;--equal signal
signal agt,bgt: std_logic;--a greater, b greater signals
signal overflow: std_logic;--overflow signals
signal tmp1,tmp2,tmp3,tmp4,tmp5,tmp6,tmp7,tmp8,tmp9,tmp10,tmp11: std_logic; --tmps
signal tmp12: std_logic_vector(2 downto 0);
--001 a>b
```

```
--010 a<b
--100 a=b

begin

    --make b in 2 comp
    s(0) <= '1'; --for 2 comp
    BtoOnesComp: for i in 0 to 7 generate
        comp: entity work.xor2(arch)
        port map  (b(i),s(0),Bcomp(i));
        --Bcomp(i)<=b(i) xor s(0); --for 1 comp
    end generate BtoOnesComp;

    --subtractor
    Subtractor: for i in 0 to 7 generate --adder thats works as a subtractor (a+(-b))
        g: entity work.FA(arch)
            port map (a(i),Bcomp(i),s(i),res(i),s(i+1));
    end generate Subtractor;

    --equals
    --y<=(res(0) nor res(1)) and (res(2) nor res(3)) and (res(4) nor res(5)) and (res(6) nor res(7));--to check if the result is zero or not
    --equals<=y and '1';
    nor1: entity work.nor2(arch)
        port map  (res(0),res(1),tmp1);        --res(0) nor res(1)
    nor2: entity work.nor2(arch)
        port map  (res(2),res(3),tmp2);        --res(2) nor res(3)
    nor3: entity work.nor2(arch)
        port map  (res(4),res(5),tmp3);       --res(4) nor res(5)
```

```
nor4: entity work.nor2(arch)
        port map  (res(6),res(7),tmp4);         --res(6) nor res(7)
and1: entity work.and2(arch)
        port map  (tmp1,tmp2,tmp5);
and2: entity work.and2(arch)
        port map  (tmp3,tmp4,tmp6);
and3: entity work.and2(arch)
        port map  (tmp5,tmp6,tmp7); --y
and4: entity work.and2(arch)
        port map  (tmp7,'1',equals); --if equals 1, if not 0

--overflow
--overflow<=s(7) xor s(8); --if overflow 1, if not 0
overf: entity work.xor2(arch)
        port map  (s(7),s(8),overflow);

--a greater than b
--agt<=(not res(7) xor overflow) and (not equals);
inv1: entity work.inv(arch)
        port map  (equals,tmp8);
inv2: entity work.inv(arch)
        port map  (res(7),tmp9);
xor1: entity work.xor2(arch)
        port map  (tmp9,overflow,tmp10);
and5: entity work.and2(arch)
        port map  (tmp10,tmp8,agt);

--b greater than a
```

```
        --bgt<=not agt and not equals;
        inv3: entity work.inv(arch)
                port map  (agt,tmp11);
        and6: entity work.and2(arch)
                port map  (tmp11,tmp8,bgt);

        --output

        tmp12 <= equals & bgt & agt;
          --output <= equals & bgt & agt;
        dff3: entity work.dff3(arch)
                port map (tmp12,clk,output);


end architecture stage1;

-------------------------------------------
--stage 2 magnitude comparator architicture
-------------------------------------------

--1 bit comparater

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--001 a>b
--010 a<b
--100 a=b
entity comp1bit is
```

```
            port ( a,b: in std_logic; eq,bgt,agt: out std_logic);
end entity comp1bit;

architecture arch of comp1bit is
signal tmp1,tmp2,tmp3,tmp4: std_logic;
begin
            --1 bit comparater
            inv1: entity work.inv(arch)
                        port map  (a,tmp1);--tmp1=!a
            inv2: entity work.inv(arch)
                        port map  (b,tmp2);--tmp2=!b
            agtb: entity work.and2(arch)--a>b
                        port map  (a,tmp2,tmp3);
            bgta:  entity work.and2(arch)--b>a
                        port map  (b,tmp1,tmp4);
            agt<=tmp3;
            bgt<=tmp4;
            equal: entity work.xnor2(arch)--b>a
                        port map  (tmp3,tmp4,eq);

end architecture arch;



--stage 2 arch
architecture stage2 of Comp8bit is
signal eq,bgta,agtb: std_logic_vector(7 downto 0);
signal equals,agt,bgt: std_logic;
signal tmp1,tmp5: std_logic_vector(3 downto 0);
signal tmp: std_logic_vector(6 downto 0);
signal tmp4: std_logic_vector(7 downto 0);
signal tmp2,tmp3,tmp6,tmp7,tmp8,tmp9: std_logic; --tmps
signal tmp10: std_logic_vector(2 downto 0);

--001 a>b
--010 b>a
--100 a=b
begin
	--equal
	--we xnor each corresponding value for a and i then and them
	Comp: for i in 0 to 7 generate --compares each bit with the corresponding one
		comp1bit: entity work.comp1bit(arch)
			port map (a(i),b(i),eq(i),bgta(i),agtb(i));
	end generate Comp;

	eq1: for i in 0 to 3 generate
		--eq(0) and eq1 = tmp1(0), eq2 and eq3 = tmp1(1), eq4 and eq5 = tmp1(2), eq6 
and eq7 = tmp1(3)
		and1: entity work.and2(arch)
		    port map  (eq(2*i),eq(2*i+1),tmp1(i));
	end generate eq1;
	and2: entity work.and2(arch)--tmp1(0) and tmp1(1)
		port map  (tmp1(0),tmp1(1),tmp2);
	and3: entity work.and2(arch)--tmp1(2) and tmp1(3)
		port map  (tmp1(2),tmp1(3),tmp3);
	and4: entity work.and2(arch)--tmp2 and tmp3
		port map  (tmp2,tmp3,equals);
```

```
        --a>b

        --
agtb(7)+e7*agtb(6)+e7*e6*agtb(5)+e7*e6*e5*agtb(4)+e7*e6*e5*e4*agtb(3)+e7*e6*e5*e4*e3*
agtb(2)+e7*e6*e5*e4*e3*e2*agtb(1)+e7*e6*e5*e4*e3*e2*e1*agtb(0)

        --which is troublesome, so will and e first then with agtb

        --note that agtb(7) is a sign bit so we use bgta(7), just for this case and this case only


        --and of eq

        --e7<=tmp6

        --e7*e6<=tmp5

        --e7*e6*e5<=tmp4

        --e7*e6*e5*e4<=tmp3

        --e7*e6*e5*e4*e3<=tmp2

        --e7*e6*e5*e4*e3*e2<=tmp1

        --e7*e6*e5*e4*e3*e2*e1<=tmp0

        tmp(6)<=eq(7);

        andeq: for i in 0 to 5 generate

                and5: entity work.and2(arch)--tmp2 and tmp3

                  port map  (tmp(6-i),eq(6-i),tmp(5-i));

        end generate andeq;


        --now i will start with the and between e and agtb equation

        --
e7*agtb(6)+e7*e6*agtb(5)+e7*e6*e5*agtb(4)+e7*e6*e5*e4*agtb(3)+e7*e6*e5*e4*e3*agtb(2)+
e7*e6*e5*e4*e3*e2*agtb(1)+e7*e6*e5*e4*e3*e2*e1*agtb(0)

        andgt: for i in 0 to 6 generate

                and6: entity work.and2(arch)

            port map  (tmp(6-i),agtb(6-i),tmp4(i));

        end generate andgt;
```

```
    --note that agtb(7) is a sign bit so we use bgta(7), just for this case and this case only
    tmp4(7)<=bgta(7);



    --now i will start with the or
    --agtb(7)+e7*agtb(6)+e7*e6*agtb(5)+e7*e6*e5*agtb(4)+e7*e6*e5*e4*agtb(3)+e7*e6*e5*e4*e3*agtb(2)+e7*e6*e5*e4*e3*e2*agtb(1)+e7*e6*e5*e4*e3*e2*e1*agtb(0)
    orgt: for i in 0 to 3 generate
        --eq(0) and eq1 = tmp1(0), eq2 and eq3 = tmp1(1), eq4 and eq5 = tmp1(2), eq6 and eq7 = tmp1(3)
        or1: entity work.or2(arch)
           port map  (tmp4(2*i),tmp4(2*i+1),tmp5(i));
    end generate orgt;


    or2: entity work.or2(arch)--tmp5(0) and tmp5(1)
        port map  (tmp5(0),tmp5(1),tmp6);
    or3: entity work.or2(arch)--tmp5(2) and tmp5(3)
        port map  (tmp5(2),tmp5(3),tmp7);
    or4: entity work.or2(arch)--tmp6 and tmp7
        port map  (tmp6,tmp7,agt);--now we got agt


    --lastly bgt
    --bgt is not agt and not equals


    inv1: entity work.inv(arch)
        port map (equals,tmp8);
    inv2: entity work.inv(arch)
        port map (agt,tmp9);
        and6: entity work.and2(arch)
                port map  (tmp8,tmp9,bgt);

        tmp10 <= equals & bgt & agt;
        --output <= equals & bgt & agt;
        dff3: entity work.dff(arch)
                port map (tmp10,clk,output);

end architecture stage2;

-----------------------------------
--verification/testbenches
-----------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE ieee.std_logic_signed.ALL;

--test generator
ENTITY TestGen IS
        PORT ( clk: IN STD_LOGIC;
        test1: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  --a
        test2: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  --b
        ExpectRes: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END ENTITY TestGen;

ARCHITECTURE arch OF TestGen IS
```

```
--001 a>b
--010 b>a
--100 a=b
BEGIN
        PROCESS
        BEGIN
                FOR i IN -128 TO 127 LOOP
                        FOR j IN -128 TO 127 LOOP
                                -- Set the inputs to the comparater
                                test1 <= CONV_STD_LOGIC_VECTOR(i,8);
                                test2 <= CONV_STD_LOGIC_VECTOR(j,8);
                                -- Calculate what the output of the comparater should be
                                WAIT until rising_edge(clk);
                                if i > j then  --a>b
                                        ExpectRes <= "001";
                                elsif j > i then --b>a
                                        ExpectRes <= "010";
                                else  --b=a
                                        ExpectRes <= "100";
                                end if;

                        END LOOP;
                END LOOP;
                WAIT;
        END PROCESS;
END ARCHITECTURE arch;

--result analyzer
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE ieee.std_logic_signed.ALL;

--result_analyzer
ENTITY resultana IS
	PORT ( clk: IN STD_LOGIC;
	ExpectRes: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
	CompRes: IN STD_LOGIC_VECTOR(2 DOWNTO 0));
END ENTITY resultana;

ARCHITECTURE arch OF resultana IS
--001 a>b
--010 b>a
--100 a=b
BEGIN
	PROCESS(clk)
	BEGIN
		IF rising_edge(clk) THEN-- Check whether output matches expectation
			ASSERT ExpectRes(2 DOWNTO 0) = CompRes
			REPORT "comparator output is incorrect"
			SEVERITY WARNING;
		END IF;
	END PROCESS;
END ARCHITECTURE arch;
```

```
------------------------------------------------
--testbench for stage1
------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_signed;

ENTITY testbenchRA IS
END ENTITY testbenchRA;

ARCHITECTURE arch OF testbenchRA IS
SIGNAL clk: std_logic:='0';
--Declarations of test inputs and outputs
SIGNAL test1: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL test2: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL compres: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL ExpectRes: STD_LOGIC_VECTOR(2 DOWNTO 0);

--001 a>b
--010 b>a
--100 a=b

BEGIN

        clk <= NOT clk AFTER 250 NS;
        -- Place one instance of test generation unit
        testgenerator: ENTITY work.TestGen(arch)
```

```
                PORT MAP ( clk, test1, test2, ExpectRes);
        -- Place one instance of the circit Under Test
        comparater: ENTITY work.comp8bit(stage1)
                PORT MAP ( test1, test2, clk, compres);
                        -- Place one instance of the result analyzer
        resultanalyzer: ENTITY work.resultana(arch)
                port map (clk, ExpectRes, compres);
END ARCHITECTURE arch;


-----------------------------------------------
--testbench for stage2
-----------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_signed;

ENTITY testbenchMA IS
END ENTITY testbenchMA;

ARCHITECTURE arch OF testbenchMA IS
SIGNAL clk: std_logic:='0';
--Declarations of test inputs and outputs
SIGNAL test1: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL test2: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL compres: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL ExpectRes: STD_LOGIC_VECTOR(2 DOWNTO 0);
```

```
--001 a>b
--010 b>a
--100 a=b

BEGIN

        clk <= NOT clk AFTER 100 NS;
        -- Place one instance of test generation unit
        testgenerator: ENTITY work.TestGen(arch)
                PORT MAP ( clk, test1, test2, ExpectRes);
        -- Place one instance of the circit Under Test
        comparater: ENTITY work.comp8bit(stage2)
                PORT MAP ( test1, test2, clk, compres);
                        -- Place one instance of the result analyzer
        resultanalyzer: ENTITY work.resultana(arch)
                port map (clk, ExpectRes, compres);

END ARCHITECTURE arch;
```